Eric Roberts and Jerry Cain
CS 106J

Handout #51
June 5, 2017

Practice Final Examination #1

Review session: **Sunday, June 11, 6:00–8:00 P.M. (Gates B-12)**

Scheduled final: **Wednesday, June 14, 8:30–11:30 A.M. (Lathrop 282)**

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the final examination. A solution set to this practice examination will be handed out on Wednesday along with a second practice exam.

Time of the exam

The final exam is scheduled Wednesday, June 14, 8:30–11:30 A.M. in the regular classroom, which means that you will have lots of space to spread out reference materials. If you are unable to take the exam at the scheduled time or if you need special accommodations, please send an email message to **`cheson@stanford.edu`** stating the following:

- The reason you cannot take the exam at the scheduled time.
- A list of three-hour blocks (or longer if you have OAE accommodations) on Monday, Tuesday, or Wednesday of exam week at which you could take the exam. These time blocks must be during the regular working day and must therefore start between 8:30 and 2:00.

In order to arrange special accommodations, Jason must receive a message from you by 5:00 P.M. on Thursday, June 8. Replies will be sent by electronic mail on Friday, June 9.

Review session

The course staff will conduct a review session on Sunday, June 11, from 6:00–8:00 P.M. in Gates B-12; your Stanford ID should let you through the curving doors facing Gilbert Biology, after which you can go down to the basement. We will announce the winners of the Adventure Contest and hold the random grand-prize drawing at the beginning of the review session.

Coverage

The exam covers the material presented in class through the class on March 2, which means that you are responsible for the material in Chapters 1 through 8 of *Understanding Programming through JavaScript.*

General instructions

The instructions that will be used for the actual final look like this:

> Answer each of the questions given below. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem. The total number of points is 100. We intend that the number of points be roughly equivalent to the number of minutes someone who is completely on top of the material would spend on that problem. Even so, we realize that some of you will still feel time pressure. If you find yourself spending a lot more time on a question than its point value suggests, you might move on to another question to make sure that you don't run out of time before you've had a chance to work on all of them.

In all questions, you may include functions or definitions that have been developed in the course, either by writing the `import` line for the appropriate package or by giving the name of the function and the handout or chapter number in which that definition appears.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Note: To conserve trees, I have cut back on answer space for the practice exams. The actual final will have much more room for your answers and for any scratch work.

Please remember that the final is open-book.

Problem 1—Short answer (10 points)

1a) Suppose that the integer array `list` has been declared and initialized as follows:

```
var list = [ 10, 20, 30, 40, 50 ];
```

This declaration sets up an array of five elements with the initial values shown in the diagram below:

`list`

10	20	30	40	50

Given this array, what is the effect of calling the function

```
mystery(list);
```

if `mystery` is defined as:

```
function mystery(array) {
   var tmp = array[array.length - 1];
   for (var i = 1; i < array.length; i++) {
      array[i] = array[i - 1];
   }
   array[0] = tmp;
}
```

Work through the function carefully and indicate your answer by filling in the boxes below to show the final contents of `list`:

`list`

1b) What is the result of calling the function `covfefe` (whatever that means) in the following code:

```
function covfefe() {
   var x = 6;
   var y = 17;
   var f = puzzle(y);
   console.log(f(x));
}

function puzzle(x) {
   return function(y) { return 2 * x - y; };
}
```

Problem 2—Simple graphics (15 points)

Although the definition of filling for an arc (see page 172 in the text) is not necessarily what you would want for all applications, it turns out to be perfect for the problem of displaying a traditional pie chart. Your job in this problem is to write a function

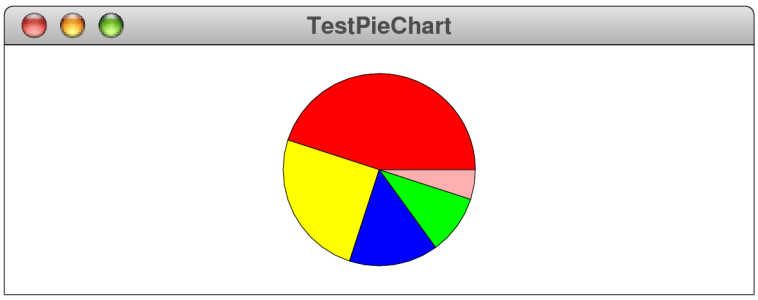
```
function createPieChart(r, data)
```

that creates a `GCompound` object for a pie chart with a set of data values, where `r` represents the radius of the circle, and `data` is the array of data values you want to plot.

The operation of the `createPieChart` function is easiest to illustrate by example. If you execute the following function:

```
function TestPieChart() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var data = [ 45, 25, 15, 10, 5 ];
   var pieChart = createPieChart(50, data);
   gw.add(pieChart, gw.getWidth() / 2, gw.getHeight() / 2);
}
```

your program should generate the following pie chart in the center of the window:



The red wedge corresponds to the 45 in the data array and extends 45% around the circle, which is not quite halfway. The yellow wedge then picks up where the red wedge left off and extends for 25% of a complete circle. The blue wedge takes up 15%, the green wedge takes up 10%, and the pink wedge the remaining 5%.

As you write your solution to this problem, you should keep the following points in mind:

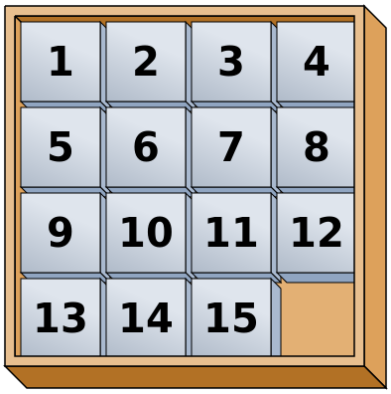
- The values in the array are not necessarily percentages. What you need to do in your implementation is to divide each data value by the sum of the elements to determine what fraction of the complete circle each value represents.
- The colors of each wedge are specified in the following constant array:

  ```
  const WEDGE_COLORS = [
     "Red", "Yellow", "Blue", "Green", "Pink", "Cyan"
  ];
  ```

  If you have more wedges than colors, you should just start the sequence over, so that the sixth wedge would be red, the seventh yellow, and so on.
- The reference point of the `GCompound` returned by `createPieChart` must be the center of the circle.

Problem 3—Interactive graphics (20 points)

In all probability, you have at some point seen the classic "Fifteen Puzzle" which first appeared in the 1880s. The puzzle consists of 15 numbered squares in a 4×4 box that looks like the following picture, which is taken from the Wikipedia entry for the puzzle:



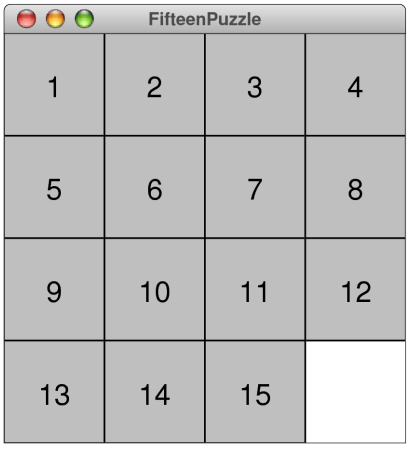
One of the squares is missing from the 4×4 grid, which creates a hole in the pattern. The puzzle is constructed so that you can slide any of the adjacent squares into the position currently taken up by the hole. The object of the game is to take a scrambled puzzle and restore it to the original state.

For this problem, your task is to create a program that simulates the Fifteen Puzzle, which is easiest to do in two steps:

Step 1.

Write a program **`FifteenPuzzle`** that displays the initial state of the Fifteen Puzzle with the 15 numbered squares arranged as shown in the diagram. Each of the pieces should be a **`GCompound`** containing a square outlined in black and filled in light gray, with a number centered in the square using an 18-point Sans-Serif font, which is given as the constant **`PUZZLE_FONT`**. The size of the square is determined by the constant **`SQSIZE`**, and the constants **`GWINDOW_WIDTH`** and **`GWINDOW_HEIGHT`** are set so the squares completely fill the graphics window.

When you have finished the code for step 1, the graphics window should look like this:

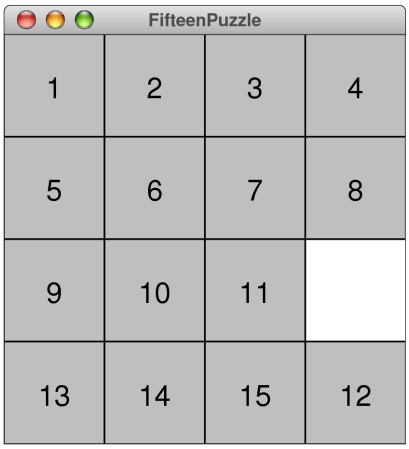


Step 2.

Animate the program so that clicking on a square moves it into the adjacent empty space, if possible. This task is easier than it sounds. All you need to do is:

1. Figure out which square you clicked on, if any, by using **`getElementAt`** to check for an object at that location.
2. Check the adjacent squares to the north, south, east, and west. If any of these squares is both inside the window and unoccupied (which can be true for at most one of the directions), move the square in that direction. If none of the directions work, do nothing.

For example, if you click on the square numbered 5 in the starting configuration, nothing should happen because all of the directions from square 5 are either occupied or outside of the window. If, however, you click on square 12, your program should figure out that there is no object to the south and then move the square to that position, as follows:



Problem 4—Strings (15 points)

In Dan Brown's best-selling novel *The Da Vinci Code,* the first clue in a long chain of puzzles is a cryptic message left by the dying curator of the Louvre. Two of the lines of that message are

O, Draconian devil!
Oh, lame saint!

Professor Robert Langdon (the hero of the book, played by Tom Hanks in the movie) soon recognizes that these lines are ***anagrams***—pairs of strings that contain exactly the same letters even if those letters are rearranged—for

Leonardo da Vinci
The Mona Lisa

Your job in this problem is to write a predicate function

```
isAnagram(s1, s2)
```

that takes two strings and returns **`true`** if they contain exactly the same alphabetic characters, even though those characters may appear in any order. Thus, your function should return **`true`** for each of the following calls:

```
isAnagram("O, Draconian devil!", "Leonardo da Vinci")
isAnagram("Oh, lame saint!", "The Mona Lisa")
isAnagram("ALGORITHMICALLY", "logarithmically")
isAnagram("Doctor Who", "Torchwood")
```

These examples illustrate two important requirements for the **`isAnagram`** function:

- The implementation should look only at letters (feel free to use the **`isLetter`** function from several textbook examples), ignoring any extraneous spaces and punctuation marks thrown in along the way.
- The implementation should ignore the case of the letters in both strings.

There are many different algorithmic strategies you could use to decide whether two strings contain the same alphabetic characters. If you're having trouble coming up with a strategy, you should note that two strings are anagrams if and only if they would generate the same letter-frequency table, as discussed in the lecture on arrays.

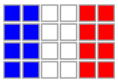
Problem 5—Arrays (10 points)

Write a function

```
function doubleImage(oldImage)
```

that takes an existing **GImage** and returns a new **GImage** that is twice as large in each dimension as the original. Each pixel in the old image should be mapped into the new image as a 2×2 square in the new image where each of the pixels in that square matches the original one.

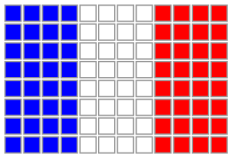
As an example, suppose that you have a **GImage** from the file **TinyFrenchFlag.png** that looks like this, where the diagram has been expanded so that you can see the individual pixels, each of which appears as a small outlined square:



This 6×4 rectangle has two columns of blue pixels, two columns of white pixels, and two columns of red pixels. Calling

```
var biggerFlag = doubleImage(GImage("TinyFrenchFlag.png"));
```

should create a new image with the following 12×8 pixel array:



The blue pixel in the upper left corner of the original has become a square of four blue pixels, the pixel to its right has become the next 2×2 square of blue pixels, and so on.

Keep in mind that your goal is to write an implementation of **doubleImage** that works with any **GImage** and not just the flag image used in this example.

I don't know if I believe in Sasquatch
but he sure do stink.

—Sherman Alexie, "The Sasquatch Poems"

Problem 6—Working with data structures (15 points)

Adventure was not the first widely played computer game in which an adventurer wandered in an underground cave. As far as we know, that honor belongs to the game "Hunt the Wumpus," which was developed by Gregory Yob in 1972.

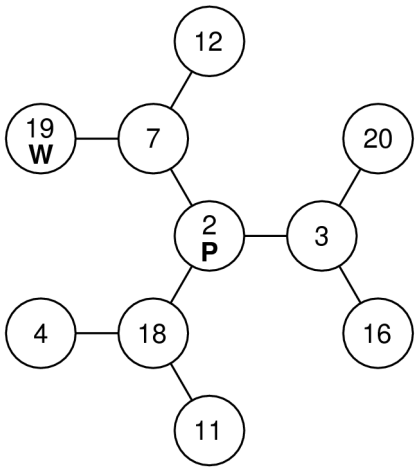
In the game, the wumpus is a fearsome beast that lives in an underground cave composed of 20 rooms, each of which is numbered between 1 and 20. Each of the twenty rooms has connections to three other rooms, represented as an array—one element per room—of three-element arrays containing the numbers of the connecting rooms. (Because the room numbers don't start with 0, the data structure puts a `null` in element 0 of the outer array, which is never used.) In addition to the connections, the data structure for the wumpus game also keeps track of the room number occupied by the player and the room number in which the wumpus resides.

In an actual implementation of the wumpus game, the information in this data structure would be generated randomly. For this problem, which focuses on whether you can work with data structures that have already been initialized, you can imagine that the data structure has been initialized using the JSON notation shown in Figure 1.

Figure 1. JSON representation of the data structure for the wumpus cave

```
const WUMPUS_CAVE = {
   playerLocation: 2,    /* The player is in room 2                 */
   wumpusLocation: 19,   /* The wumpus is in room 19                */
   connections: [
      null,              /* Room 0 is not used                      */
      [6, 14, 16],       /* Room 1 connects to rooms 6, 14, and 16  */
      [3, 7, 18],        /* Room 2 connects to rooms 3, 7, and 18   */
      [2, 16, 20],       /* Room 3 connects to rooms 2, 16, and 20  */
      [6, 18, 19],       /* Room 4 connects to rooms 6, 18, and 19  */
      [8, 9, 11],        /* Room 5 connects to rooms 8, 9, and 11   */
      [1, 4, 15],        /* Room 6 connects to rooms 1, 4, and 15   */
      [2, 12, 19],       /* Room 7 connects to rooms 2, 12, and 19  */
      [5, 10, 13],       /* Room 8 connects to rooms 5, 10, and 13  */
      [5, 11, 17],       /* Room 9 connects to rooms 5, 11, and 17  */
      [8, 14, 16],       /* Room 10 connects to rooms 8, 14, and 16 */
      [5, 9, 18],        /* Room 11 connects to rooms 5, 9, and 18  */
      [7, 14, 15],       /* Room 12 connects to rooms 7, 14, and 15 */
      [8, 15, 20],       /* Room 13 connects to rooms 8, 15, and 20 */
      [1, 10, 12],       /* Room 14 connects to rooms 1, 10, and 12 */
      [6, 12, 13],       /* Room 15 connects to rooms 6, 12, and 13 */
      [1, 3, 10],        /* Room 16 connects to rooms 1, 3, and 10  */
      [9, 19, 20],       /* Room 17 connects to rooms 9, 19, and 20 */
      [2, 4, 11],        /* Room 18 connects to rooms 2, 4, and 11  */
      [4, 7, 17],        /* Room 19 connects to rooms 4, 7, and 17  */
      [3, 13, 17]        /* Room 20 connects to rooms 3, 13, and 17 */
   ]
};
```

Looking at the data structure allows you to diagram the rooms in the cave. Here is a piece of the cave map centered on the location of the player in room 2:



The player is in room 2, which has connections to rooms 3, 7, and 18. Similarly room 7 has connections to rooms 2, 12, and 19, which is where the wumpus is lurking. The other connections from rooms 4, 11, 16, 20, 12, and 19 are not shown.

It was usually possible to avoid the wumpus because, like the Sasquatch in the Sherman Alexie poem, the wumpus was so stinky that the player could smell the wumpus from up to two rooms away. Thus, in the diagram above, the player can smell the wumpus. If, however, the wumpus were to wake up and move to a room beyond the boundaries of this diagram, the scent of the wumpus would disappear.

Write a predicate function **`doesPlayerSmellTheWumpus`**, which takes the entire data structure as an argument and returns **`true`** if the player smells the wumpus and **`false`** otherwise. Thus, calling

```
doesPlayerSmellTheWumpus(WUMPUS_CAVE)
```

would return **`true`**. The function would also return **`true`** if the wumpus were in rooms 3, 7, or 18, which are one room away from the player. If, however, the wumpus were in one of the rooms not shown on this map, **`doesPlayerSmellTheWumpus`** would return **`false`**.

Problem 7—Reading data structures from files (15 points)

One of the political events of interest in the next week is the British general election scheduled for June 8. In each parliamentary district, which are called ***constituencies*** in Britain, several candidates from different parties will be running, although not all parties will contest every election. Each candidate will receive some number of votes in the election. Under Britain's "first past the post" system, the candidate who wins the greatest number of votes wins the seat for that constituency, even if that candidate does not win a majority of the votes cast.

Imagine that you have been hired by a consulting firm that seeks to analyze the results of the election, which have been delivered in a large file that looks like this, which shows the results of the 2015 general election:

BritishElectionData.txt

```
Aberavon
Stephen Kinnock (Labour) 15416
Peter Bush (UKIP) 4971
Edward Yi He (Conservative) 3742

Aberconwy
Guto Bebb (Conservative) 12513
Mary Wimbury (Labour) 8514
Andrew Haigh (UKIP) 3467

Aberdeen North
Kirsty Blackman (SNP) 24793
Richard Baker (Labour) 11397
Sanjoy Sen (Conservative) 5304
Euan Davidson (LibDem) 2050
  .
  .
  .
Brighton Pavilion
Caroline Lucas (Green) 22871
Purna Sen (Labour) 14904
Clarence Mitchell (Conservative) 12448
  .
  .
  .
York Outer
Julian Sturdy (Conservative) 26477
Joe Riches (Labour) 13348
James Blanchard (LibDem) 6269
Paul Abbott (UKIP) 5251
```

Each entry in the file consists of the name of the constituency on the first line, followed by as many lines as there were candidates in that constituency. The end of the candidate list is marked with a blank line or the end of the file. Each candidate line consists of the candidate name, the party name in parentheses, and the number of votes cast for that candidate. You may assume that the file exists and is properly formatted, which means that you don't have to include any error-checking.

Implement a JavaScript **`ElectionData`** class whose factory method has the following header line:

```
function ElectionData(filename)
```

Calling **ElectionData** should read the contents of the file and returns an initialized data structure that includes the data in a suitable internal form. Your data structure for **ElectionData** should include the following methods:

- A method **getConstituencyNames** that returns an array containing the constituency names that appear in the file.
- A method **getResults(name)** that returns the results of the election for the constituency with the specified name. The result is an array, each of whose elements is an aggregate with the fields **candidate**, **party**, and **votes**. For example, calling **getResults("Aberavon")** should return the following array, as it would be expressed in JSON form:

```
[
  { candidate:"Stephen Kinnock", party:"Labour", votes:15416 },
  { candidate:"Peter Bush", party:"UKIP", votes:4971 },
  { candidate:"Edward Yi He", party:"Conservative", votes:3742 }
]
```

In this problem, all you have to do is read the data into the internal structure. Any actual analyses of results are the responsibility of the clients of your **ElectionData** class. For example, if someone wanted to determine the total vote for the Labour party across all constituencies, that person could use the following code:

```
function TestElectionData() {
   var electionData = ElectionData("BritishElectionData.txt");
   var totalLabourVote = 0;
   var constituencies = electionData.getConstituencyNames();
   for (var i = 0; i < constituencies.length; i++) {
      var results = electionData.getResults(constituencies[i]);
      for (var j = 0; j < results.length; j++) {
         if (results[j].party === "Labour") {
            totalLabourVote += results[j].votes;
         }
      }
   }
   console.log("Total Labour vote = " + totalLabourVote);
}
```